ECSE 2920: Design Methodology

Deliverable #6

Group 11

02/06/2025

Part 1: Dip Switch Mode inputs

Requirements: Our team was tasked with implementing DIP switches to enable different mode inputs for the audio car.

The table below shows how the DIP switches control each mode, including turning the car off or switching between modes like “Straight”, “Figure 8”, and “Square.”
The car needs to function without a monitor or manual control, requiring auto-start configuration. Additionally, the design should light up LEDs corresponding to the state of each switch to provide visual feedback during operation. The table provided for switch behavior is as follows:

Bit 3	Bit 2	Bit 1	Bit 0	Mode
0	x	x	x	Don’t Move
1	0	0	0	Single
1	0	0	1	Multiple
1	1	0	0	Straight
1	1	0	1	Figure 8
1	1	1	0	Square
1	1	1	1	Debug

Picture(s)

GPIO Pin Occupations

Board Pin	GPIO Pin	Purpose
11	GPIO17	Left Motor A
12	GPIO18	Left Motor B
31	GPIO6	Right Motor A
32	GPIO12	Right Motor B
13	GPIO27	Dipswitch Bit 0
15	GPIO22	Dipswitch Bit 1
38	GPIO20	Dipswitch Bit 2
40	GPIO21	Dipswitch Bit 3
22	GPIO25	GREEN LED (Bit 0)
23	GPIO11	WHITE LED (Bit 1)
21	GPIO9	YELLOW LED (Bit 2)
19	GPIO10	RED LED (Bit 3)

(***Figure 1:*** *Verified excel sheet showing GPIO Pin occupations*)

```
# Mode dictionary that maps bit strings to commands
commandArray = {
    "0000": "Don't Move",
    "1000": "Single",
    "1001": "Multiple",
    "1100": "Straight",
    "1101": "Figure 8",
    "1110": "Square",
    "1111": "Debug"
}

# Read and process switch values
bitString = ""
for i in range(4):
    if gpioArray[i].is_pressed:
        bitArray[3 - i] = 1
    else:
        bitArray[3 - i] = 0
    bitString += str(bitArray[i])

# Check if mode exists in commandArray and print it
if bitString in commandArray:
    print("Command: " + commandArray[bitString])
```

(***Figure 2:*** *code segment showing use of commandArray dictionary*)

```
from gpiozero import Button, LED

# Define switches and LEDs
bit0 = Button(27, pull_up=False, bounce_time=0.5)
bit1 = Button(22, pull_up=False, bounce_time=0.5)
bit2 = Button(20, pull_up=False, bounce_time=0.5)
bit3 = Button(21, pull_up=False, bounce_time=0.5)

led0 = LED(25)
led1 = LED(11)
led2 = LED(9)
led3 = LED(10)

# Arrays to organize GPIOs and mode detection
gpioArray = [bit0, bit1, bit2, bit3]
ledArray = [led0, led1, led2, led3]
bitArray = [0, 0, 0, 0]
```

(***Figure 3:*** *code snippet showing how we defined and initialized the switches and LEDs*)

Key Design Decision(s)

- What did your team clarify about the design?
    - We clarified how to assign GPIO pins for each DIP switch and corresponding LED. Initially we encountered issues where switches were lighting the wrong LEDs due to GPIO number mismatches. Through testing, we verified the pin connections and corrected the code. We also confirmed the use of pull-up resistors and debounce timing to prevent false readings from switch bounce
    - Additionally, we discussed how to implement mode switching in the program. Since each mode activates only when Bit 3 is set, we designed the code to continuously monitor the DIP switches for changes and ensure that switching modes dynamically lights up the correct LEDs.
- What were the competing choices in the design?
    - GPIO pin assignment: we debated the optimal GPIO pins to use for the DIP switches and LEDs to minimize potential interference and ensure ease of wiring on the breadboard
    - Handling debounce and hysteresis:
        - We discussed whether to implement a more advanced software hysteresis algorithm to ensure smooth mode switching. Ultimately, we decided that the current debounce time (bounce_time = 0.5) was sufficient and did not require additional tweaking since testing showed stable performance.
        - Our physical form of hysteresis/redundancy is how power if supplied to the dipswitches. Power is passed through switch 1 (our bit 3) to the other switches, so if bit 3 is off, then it's impossible for the other bits to receive power.
        - We also have a software form of hysteresis as well. The car sits on idle for up to 2 minutes when all bits are zero. Once the switches change to a recognized command value, the switch values must stay static for 7 seconds before the car starts the command.
- Ultimately what did your team choose and why?
    - We chose to stick with the current debounce and GPIO configuration because the design performed reliably during testing. After encountering issues where DIP switches activated the incorrect LEDs, we re-evaluated and verified our GPIO assignments in an excel spreadsheet, ensuring that each switch and LED was mapped correctly. This is shown in ***figure #1***.
    - We also maintained a debounce time of 0.5 seconds to filter out switch bounce, ensuring stable and accurate readings without false triggers. By organizing the GPIOs systematically and verifying the LED output logic, we achieved reliable mode activation and visual feedback that met the project requirements. This setup was further validated by our auto start tests, where the DIP switches and LEDs functioned correctly upon powering the RP4.
    - To handle mode selection, we used a commandArray dictionary to map each combination of DIP switch values to a corresponding mode. The program

continuously reads the state of all four switches, generated a bit string representing their current state, and uses this string to look up the mode in the dictionary. If a match is found, the program prints the corresponding mode and activates it when Bit 3 (Go/No-Go) is set to 1. ***Figure #2*** is a code snippet illustrating this logic.

TEST… Test… test

- What aspects of the design need to be tested?
    - Software?
        - We developed and tested our program using Python and the gpiozero library. The key components tested included GPIO detection and LED outputs. ***Figure #3*** is a code snippet showing how we defined and initialized the switches and LEDs
    - Hardware?
        - We tested the circuit on a breadboard, ensuring each switch correctly updated the program's mode and that the LEDs provided accurate feedback. Additionally, we confirmed that the system booted and ran the program automatically without requiring a monitor or manual input.
- Who is responsible for testing?
    - Tests ran?
        1. We tested each mode by toggling the DIP switches and observing LED outputs to verify correctness
        2. We confirmed that the car's program auto started and monitored DIP switch changes after being powered on.
        3. We confirmed that Bit 3 must be switched off to enable mode changes and that switching it on activated the selected mode.
    - Conclusions from testing?
        - Testing verified that the DIP switches and LEDs function as expected. We successfully implemented mode switching and GPIO detection with visual feedback from LEDs. The debounce setting proved adequate, and the program worked reliably on auto start.

summary/part conclusion (make sure you address all parts of the requirements)

For this part, we implemented DIP switch inputs and LED feedback for the audio car's mode selection. We verified the correct wiring and GPIO configuration, fixed initial pin assignment issues, and confirmed stable performance with debounce handling. The program auto starts when the Pi is powered on, fulfilling all design requirements. We are now prepared to implement the mode-specific drawing functions for further testing and demonstration.

Part 2: Figure Drawings

Requirements: Have proof of and be able to demonstrate physical drawings, the audio car should be able to go in a straight line, a square, and a figure eight.

***Video is uploaded which displays the audio car completing all three physical drawings**

Key Design Decision(s)

- What did your team clarify about the design?
    - How are we going to implement/modify our transfer function?
    - How will our audio car make turns?
    - How will we drive the figure 8?
- What were the competing choices in the design?
    - For the transfer function:
        - Develop into PID
        - Keep transfer function
        - Switch to Hard Code and trial/error
    - For making turns:
        - Turn both wheels opposite directions
        - Turn only one wheel to turn
    - Figure 8:
        - Full Arc (Hardest)
        - Slight turn, smaller arc, then turn again
        - Collection of small straights and turns. No arcs (Least Complicated)
- Ultimately what did your team choose and why?
  We decided to redo a transfer function by gathering our data experimentally. This involved having the car on the demo carpet and modifying duty cycle values until we got a ratio between both motors that resulted in straight driving. During this modification, we discovered that the motors would have different rotations and offset start times as well. In our case, the right motor started earlier and rotated faster, so we had to modify the driveControls program to account for this.

  We originally had the car turn by rotating both wheels in opposite directions. Unfortunately, this led to extremely inaccurate and inconsistent results, even though we could already have the robot drive straight. Once we switched to having the car turn by driving a single wheel, we were able to marginally increase both predictability and accuracy within the car's behavior.

For making the figure 8 turns, we decided to do the second option where we make a slight turn, arc, and turn again. It's a happy medium between the extreme options that presents an impressive, yet convenient result that didn't take extremely long to perfect.

TEST… Test… test

- What aspects of the design need to be tested?

    - Software?
        - The dipswitch program (i.e. the main program) needs to call the functions inside driveControls.py, which contains the written scripts for driving maneuvers and completing each figure.
    - Hardware?
        - This section is particularly about the interaction between hardware and software. Rigorous testing needs to be done in each shape, and most importantly, that testing should be on the grounds where the demonstration is being done. Everything we have up to now needs to be tested. This includes the dipswitch commands, moving straight, turning both directions with one wheel, and moving the car in an arc.
- Who is responsible for testing?
    - Tests ran?
        - Over multiple days, we tested each movement in the spare hours that the power auditorium offered. We made sure to test it as if we were doing the demo. This means that the commands were called on startup while testing, which got us used to the car's behavior on startup as well.
    - Conclusions from testing?
        - The car calls the correct figure drawings on startup while traversing them with acceptable consistency. As the figures increase in complexity, more things can go wrong, and it seems like the straight line is the most consistent, with the square not too far behind. Our figure eight CAN work, but with the amount of turning it does along with the tension from the cables, it is the most inconsistent out of the three. It requires near perfect handling from the "walker" to make sure that the cables don't pull and redirect the car.

summary/part conclusion (make sure you address all parts of the requirements)

The audio car successfully demonstrated movement in a straight line, a square, and a figure-eight, with video proof uploaded. To achieve this, we refined our transfer function through experimental data, adjusting motor speeds for accuracy. Turning both wheels in opposite directions was inconsistent, so we switched to single-wheel turns, improving precision. For the figure-eight, we chose a slight turn and arc method for a balance of complexity and control. Testing confirmed that the car reliably follows the programmed paths, though the figure-eight remains the most inconsistent due to cable tension.

Conclusion and Participation (REQUIRED)

1. **Include a selfie/photo from your group meeting/zoom call.**

2. **When did you meet? 2/10/2025 & 2/11/2025**
3. **Who was present? Dawson Gulasa, Finn Morton, Michael See, Cade Toney**
4. **Who was not present? Nobody was absent both days**
5. **What were the main ideas discussed or major decisions (1-2 sentences/bullet points)**
    - **Dipswitch operations and how to make them demo-friendly**
    - **How to navigate the figure 8**